SYSTEM AND METHOD OF ERROR RETENTION FOR THREADED SOFTWARE

FIELD OF THE INVENTION

In general, the invention relates to computer software. More specifically, the invention relates to increased error specification using a method of error retention for threaded production software.

BACKGROUND OF THE INVENTION

Defects in production application software that process a large volume of requests via frequently created short-lived threads, such as applications implemented on modern Web application servers, are notoriously difficult to correct.

Development teams that build application software typically implement a configurable logging facility that produces log statements that are persisted to a file, or to a database. These logging facilities usually allow various levels of logging. At debug levels, the typical logging facility produces a detailed log. At production levels, typically only a health check or detected error conditions are logged.

The statements captured in a debug level log are invaluable when diagnosing the cause of unexpected errors especially those caused by defects in the software itself. However, if production software were to log debug level statements, the resulting performance degradation caused by persisting the log would be prohibitive. As production software executes more threads, the multiple threads would exacerbate the performance degradation.

Thus, there is a significant need for a system and method for improving retention of error information (error logging) that overcomes the above disadvantages and shortcomings, as well as other disadvantages.

BRIEF SUMMARY OF THE INVENTION

One aspect of the invention provides a method for error detection of multi-threaded application software. The method may include providing application threads with access to a logger that collects a thread's log statements in memory and can persist all of those statements in case of an error in the production environment. The method may also include detecting a death of an application thread by the logger and deleting the application thread's log statements after thread death detection.

Another aspect of the invention provides a system for error detection of multi-threaded software. The system comprising a means for executing an application which uses a logger that collects log statements, means for collecting at least one log statement from at least one application thread and storing the at least one log statement in memory and means for allowing the collected log statement to be persisted in case of an error in a production environment. The system may also include means for detecting a death of an application thread by the logger and means for deleting the application thread's log statements after thread death detection.

Another aspect of the invention provides a computer readable medium storing a computer program. The computer readable medium comprising computer readable code for executing an application which uses a logger that collects log statements, computer readable code for collecting at least one log statement from at least one application thread and storing the at least one log statement in memory, and computer readable code for allowing the collected log statement to be persisted in case of an error in a production environment. The computer readable medium may include computer readable code for detecting a death of an application thread by the logger, and computer readable code for deleting the application thread's log statements after thread death detection.

The foregoing and other features and advantages of the invention will become further apparent from the following detailed description of the presently preferred embodiment, read in conjunction with the accompanying drawings. The detailed description and drawings are merely illustrative of the invention rather than limiting, the scope of the invention being defined by the appended claims and equivalents thereof.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram illustrating one embodiment for a method of operation of an in-memory logger in accordance with the present invention;

FIG. 2 is a flowchart representation of one embodiment for an application thread interaction with the in-memory logger of **FIG. 1**, in accordance with the present invention;

FIG. 3 is a flowchart representation of a preferred embodiment for an instance method used by the application thread of **FIG. 2**, in accordance with the present invention;

FIG. 4 is a flowchart representation of a preferred embodiment for a constructor of a logger object as associated with the instance method of **FIG. 3**, in accordance with the present invention;

FIG. 5 is a flowchart representation of a preferred embodiment for a configuration monitor as associated with the logger object of **FIG. 4**, in accordance with the present invention;

FIG. 6 is a flowchart representation of a preferred embodiment for a thread death monitor as associated with the logger object of **FIG. 4**, in accordance with the present invention;

FIG. 7 is a flowchart representation of a preferred embodiment for a log message thread, as associated with the logger object of **FIG. 4**, in accordance with the present invention;

FIG. 8 is a flowchart representation of a preferred embodiment for the destructor of the logger object of **FIG. 4**, in accordance with the present invention;

FIG. 9 is a flowchart representation of a preferred embodiment for a log message thread, as associated with the logger object of **FIG. 4**, in accordance with the present invention; and

FIG. 10 is a flowchart representation of a preferred embodiment for a write log method, used by the application thread of **FIG. 2**, in accordance with the present invention.

DETAILED DESCRIPTION OF THE PRESENTLY PREFERRED EMBODIMENTS

FIG. 1 illustrates one embodiment for a method of operation of an in-memory logger **100** in accordance with the present invention. The illustrated embodiment of the invention demonstrates that a logger **120** may reside on an operating system **150**, and an additional embodiment may provide the operating system **150** to reside on a Web Application Server **140**. The logger **120** may respond to a request from threads of execution operating as part of a software application **110**. One embodiment of the invention may depict the interacting software application **110** as a Web application. Additionally, **FIG. 1** illustrates an embodiment of the invention where the logger **120** may be integrated into the base class of an application framework **130** thus easing the applications' **110** access to the logger.

FIG. 2 illustrates, as a flowchart representation, one embodiment of how an application thread may interact with an embodiment of an in-memory logger (logger) compatible with the system of **FIG. 1**. An application (production software) may activate a thread **205**. The application thread may obtain an instance of the logger **210**. In one embodiment of the invention the logger may be a Singleton design pattern object. The logger can then receive data (log statements) from the application thread **215, 220**. For one embodiment of the

invention, the application thread can indicate to the logger the log statement's retention level **215, 200**. If the application thread detects an error, the application thread can call the logger's write log method to direct the logger to persist all log statements the logger received from the application thread **225**. One embodiment of the invention may place each log statement into a log message queue, store the log statements in memory, and may write the log statements to a persistence mechanism as is known in the art. After the production software's application thread termination, the logger will detect the thread's death and discard the log statements stored in memory **230**.

Illustrated in **FIG. 3**, is an embodiment of the logger as a Singleton design pattern object **300**. When an instance of the logger is requested **310**, the logger's static instance method may inquire a static variable indicating whether an instance of the logger has already been constructed **320**. If the logger has not yet been constructed then the instance method will construct a new logger object **330**. Upon exit of the instance method, the single instance of the logger object is returned **340**.

FIG. 4 illustrates a flowchart representation of a logger object's constructor **400** associated with the logger's instance method **300** of **FIG 3**. When activated **405**, the logger object constructor **400** may begin by instantiating an object to inventory the threads **415**. In addition, the logger object constructor **400** may instantiate an object to manage any accumulated application threads' log statements **425**. A thread death monitor may be activated **435** to monitor application threads' operational status. The minimum retention level of qualifying log statements and a predetermined waiting period for monitoring function control is retrieved **445** by the logger object constructor **400** for use with the activation of a configuration monitor **455**. The logger object constructor **400** may continue by instantiating a log message queue **465** and complete operation by activating a log message thread to monitor the log message queue **475**.

One embodiment of the configuration monitor associated with the logger object constructor **400** of **FIG. 4** is illustrated as a flowchart in **FIG. 5**. After its activation **505**, the configuration monitor **500** may suspend operation **510** for a duration of time (waiting period) defined in the logger object constructor **400** of **FIG. 4**. After the waiting period has lapsed, the configuration monitor **500** looks for a termination indicator. If a termination indicator is found then the configuration monitor may exit **540**. If a termination indicator is not found then the configuration monitor may re-obtain the minimum retention level of qualifying log statements and a predetermined waiting period for monitoring function and configuration data **530** and may again suspend operation **510**.

One embodiment of the thread death monitor associated with the logger object constructor **400** of **FIG. 4** is illustrated as a flowchart in **FIG. 6**. After its activation **605**, the thread death monitor **600** suspends operation **610** for the duration of time defined in the logger object constructor **400** of **FIG. 4** or redefined in the configuration monitor **500** of **FIG. 5**. After the waiting period has lapsed, the thread death monitor **600** looks for a termination indicator **615**. If a termination indicator is found then the configuration monitor may exit **640**. Otherwise, for each application thread that requested an instance of the logger **620**, the thread death monitor **600** obtains the application thread's state of operation **625**. If an application thread is found to be inactive, the thread death monitor **600** may remove the application thread's log from the log manager **630** and may remove the application thread from the thread manager **635**. After processing each application thread entry in the thread manager **620**, the thread death monitor **600** may again suspend operation **610**.

FIG. 7 illustrates, as a flowchart, the operations of the logger's log method **700** that is exposed to application threads to allow application threads to pass a log statement to the logger. Activation of the log method **700** occurs when an application thread submits a log statement to the logger along with the log statement's retention level **705**. The log method **700** may then obtain a time stamp **715**. The log method **700** may continue and obtain the requesting application thread's identification **720**. The application thread's identification data is compared to thread identification data in the thread manager **725**.

If the application thread's identification is not found in the thread manager, the log method **730** may check if application thread's identification is unique. If the application thread's identification is not unique then the log method **700** may assign unique thread identification **735** to the application thread. Once the unique thread identification has been assigned, the application thread's unique thread identification may be added to the thread manager **740**.

Once the application thread's identification is in the thread manager, the log method **700** may combine the timestamp, the application thread's identification, and log statement as a new log message **745**. The log method **700** may then write the new log message along with its retention level to the log message queue **750**, which ends the log method **700** operation.

FIG. 8 illustrates a flowchart representation of a logger object destructor **800**. When activated **810**, the logger object destructor **800** may begin by setting a termination indicator to true **820** which may cause the configuration monitor **500** and the thread death monitor **600** to terminate. The logger object destructor may then use the log method to pass a log statement to background log message thread indicating that it too should terminate **830** which ends the logger object destructor **800** operation.

FIG. 9 illustrates one embodiment of a log message thread **900** associated with the logger object constructor **400** of **FIG. 4**. The log message thread processes messages placed on the logger's log message queue. Subsequent to activation **910**, the log message thread may repeatedly obtain and process messages arriving on the log message queue **920**. For each arriving message, the log message thread may dequeue the log message and its retention level from the log message queue **920**. If the retention level has been set to equal terminate **930**, the log message thread **900** may terminate **970**. Otherwise, the error message may be added to the application thread's log in the log manager **940**. If the log message's retention level is greater then or equal to the minimum retention level defined in the logger object constructor **400** of **FIG. 4** or redefined in the configuration monitor **500** of **FIG. 5**, the log message thread may write the log message to the persistence mechanism **960**. After the message has been written **960**, or if the retention level is less then the minimum retention level **950**, the log message thread **900** may return to obtain process another log message from the log message queue **920**. This process may continue until the retention level has been set to equal terminate **930**, and the log message thread **900** terminates **970**.

The logger's write log method **984**, illustrated in **FIG. 10** as a flowchart and associated with **FIG. 1**, writes the requesting application thread's log statements to the persistence mechanism. The method may begin **984** by obtaining the requesting application thread's identification **988**. The write log method **984** may retrieve the requesting application thread's log statements from the log manager **990**. The write log method **984** may then be written to the retrieved log statements to the persistence mechanism **994**, which then ends the write log method **984** operation.

The above-described methods and implementation of error retention for threaded production software are example methods and implementations. These methods and implementations illustrate one possible approach for error retention for threaded production software. The actual implementation may vary from the method discussed. Moreover, various other improvements and modifications to this invention may occur to those skilled in the art, and those improvements and modifications will fall within the scope of this invention as set forth below.

While the embodiments of the invention disclosed herein are presently considered to be preferred, various changes and modifications can be made without departing from the spirit and scope of the invention. The scope of the invention is indicated in the appended claims, and all changes that come within the meaning and range of equivalents are intended to be embraced therein.